



Sistemi
Operativi

Bruschi
Monga

Software
factory

Make
Debugger
Low level
programming
diff & patch
Versioning

Sistemi Operativi¹

Mattia Monga

Dip. di Informatica
Università degli Studi di Milano, Italia
mattia.monga@unimi.it

a.a. 2012/13

¹ © 2011–13 M. Monga. Creative Commons Attribution-Condividi allo stesso modo 3.0 Italia License.
<http://creativecommons.org/licenses/by-sa/3.0/it/>. Immagini tratte da [?] e da Wikipedia.



Sistemi
Operativi

Bruschi
Monga

Lezione XIX: The UNIX software factory

Software
factory

Make
Debugger
Low level
programming
diff & patch
Versioning



- UNIX nasce come sistema *per i programmatori* (l'unica tipologia di utente all'inizio degli anni '70...)
- progettato insieme ad un linguaggio di programmazione (C)
- la 'filosofia di UNIX' (piccoli programmi che fanno molto bene una sola cosa su file) si adatta perfettamente al paradigma di sviluppo **edit-compile-debug**
- tool all'avanguardia nell'elaborazione di *file di testo* (per lo piú organizzati per "righe") e per la scrittura dei programmi di elaborazione stessi (lex, yacc,...)



- Editor: `ed`, `vi`, `emacs` manipolano arbitrariamente i byte di un file, generalmente interpretandoli come caratteri stampabili (testo)
- Compilatore: `cc` (`gcc`)
 - 1 `cc` sorgente (`.c`) \rightsquigarrow assembly (`.s`)
 - 2 `as` assembly \rightsquigarrow *oggetto* (`.o`)
 - 3 (`ar` archivia diversi oggetti in una *libreria* (`.a`))
 - 4 `ld` *oggetti* e *librerie* \rightsquigarrow eseguibile (`a.out`) (il formato storico è COFF, oggi ELF)

Si noti che a sua volta anche la compilazione vera e propria è fatta da due passi (pre-processor `cpp` e compilazione `cc1`).



Stuart Feldman, 1977 at Bell Labs.

Permette di specificare **dipendenze** fra processi di generazione.

Dipendenze: se cambia (secondo la data dell'ultima modifica) un prerequisito, allora il processo di generazione deve essere ripetuto.

```
1 helloworld.o: helloworld.c
2         cc -c -o helloworld helloworld.c
3
4 helloworld: helloworld.o
5         cc -o $@ $<
6
7 .PHONY: clean
8 clean:
9         rm helloworld.o helloworld
```



Breakpoint

Un punto del programma in cui l'esecuzione deve essere bloccata, tipicamente per esaminare lo stato in quell'istante.

Stepping

Eseguire il programma *passo a passo*. La granularità del passo può arrivare fino all'istruzione macchina.

Software
factory

Make
Debugger
Low level
programming
diff & patch
Versioning



Lo stato del programma può essere analizzato come:

- **forma simbolica**: secondo i simboli definiti nel linguaggio di alto livello e conservati come *simboli di debugging*
- **memoria virtuale**: stream di byte suddiviso in segmenti
 - Text: contiene le istruzioni (spesso read only)
 - Initialized Data Segment: variabili globali inizializzate
 - Uninitialized Data Segment (bss): variabili globali non inizializzate
 - Stack: collezione di *stack frame* per le chiamate di procedura. Cresce verso il basso.
 - Heap: Strutture dati create dinamicamente. Cresce verso l'alto tramite system call `brk` (API `malloc`).



- `break ...` (un simbolo o un indirizzo `*0x...`)
- `run ...` (eventualmente con `argv`)
- `print ... (x)`
- `next (nexti)`
- `step (stepi)`
- `backtrace`



La *symbol table* serve al *linker* per associare nomi simbolici e indirizzi prodotti dal compilatore:

- contenuta in tutti gli oggetti, generalmente viene lasciata anche negli eseguibili (ma può essere scartata con `strip`)
- una versione piú ricca viene detta “simboli di debug” (vari formati, p.es. DWARF)
- le tabelle dei simboli possono essere consultate con `nm`



Perché capire i dettagli delle fasi?

Per costruire sistemi operativi a volte serve alterare il flusso tradizionale

```
1 gcc -O -nostdinc -l. -c bootmain.c
2 gcc -nostdinc -l. -c bootasm.S
3 ld -m elf_i386 -N -e start -Ttext 0x7C00 -o bootblock.o bootasm.o bootmain.o
4 objdump -S bootblock.o > bootblock.asm
5 objcopy -S -O binary -j .text bootblock.o bootblock
```

```
1 $ nm kernel | grep _start
2 8010b50c D _binary_entryother_start
3 8010b4e0 D _binary_initcode_start
4 0010000c T _start
```

Sistemi
Operativi

Bruschi
Monga

Software
factory

Make
Debugger
Low level
programming
diff & patch
Versioning



Assembly in C

Sistemi
Operativi

Bruschi
Monga

In alcuni casi è comodo mischiare l'assembly al C (meno laborioso di organizzare il collegamento)

Software
factory
Make
Debugger
Low level
programming
diff & patch
Versioning

```
1 __asm__("nop");  
2  
3 __asm__("movl %eax, %ebx");  
4 __asm__("xorl %ebx, %edx");  
5 __asm__("movl $0, _booga");  
6  
7 __asm__("pushl %eax\n\t"  
8         "movl $0, %eax\n\t"  
9         "popl %eax");
```

Attenzione! Il compilatore C non "vede" l'effetto delle istruzioni assembly.



Assembly in C (cont.)

Si possono fare anche cose piú complicate, ma la sintassi è poco “amichevole”

```
1 __asm__("cld\n\t"  
2      "rep\n\t"  
3      "stosl"  
4      : /* no output registers */  
5      : "c" (count), "a" (fill_value), "D" (dest)  
6      : "%ecx", "%edi" );
```

La sintassi è

```
1 __asm__( "statements" : output_registers : input_registers : clobbered_registers);
```

http://www.delorie.com/djgpp/doc/brennan/brennan_att_inline_djgpp.html

Sistemi
Operativi

Bruschi
Monga

Software
factory

Make
Debugger
Low level
programming
diff & patch
Versioning



Con `cmp` è possibile controllare se due file sono identici.
Per i file di testo organizzato il righe esistono strumenti più sofisticati:

- `diff` elenca le modifiche necessarie per trasformare un file in un altro (`diff3` si aiuta con un “antenato” comune, fondamentale per facilitare il *merge*)
- `diff` (e in maniera più evoluta `diff3`) cerca di identificare le righe che *non sono cambiate*: le modifiche sono organizzate per **hunk**
- `patch` riapplica gli hunk di modifica al file originale (o versioni *leggermente* modificate dei medesimi)



Dagli anni '80 sono stati proposti molti strumenti per trattare in modo efficiente:

- le successive revisioni di un file
- le versioni di un prodotto software
- le configurazioni che permettono di ottenere una specifica versione del prodotto

SCCS, RCS, CVS, SVN, git...

Si basano tutti sulla conservazione della “storia” dello sviluppo in un *repository*: per lavorare occorre fare *checkout* di un *artifact*, e poi chiedere il *commit* delle modifiche.



L'idea può essere incorporata a vari livelli: Emacs può “salvare” automaticamente le versioni precedenti dei file (generalmente una sola *, altrimenti * 1 ...), oppure addirittura nel *file system*.

Git invece ricrea un suo “file system”: **blob** e **tree**, **ref**.

- multi-phase commit: *working directory*, *stage* e *local repository*
- distribuito senza necessariamente server centralizzati: pull e push
- in un **commit** è conservato l'insieme delle modifiche (come 'diff') fatte ad un insieme (*change-set*) di file: perciò è associato a un *tree*
- una **branch** è semplicemente una *reference* mobile a una linea di sviluppo.